C++学习笔记5

2022年3月19日 20:49

- IO类
  - 为了支持不同的IO操作，在istream和ostream之外，标准库还定义了一些其他的IO类型
    - iostream定义了用于读写流的基本类型，fstream定义了读写命名文件的类型，sstream定义了读写内存string对象的类型

表 8.1：IO 库类型和头文件

头文件	类型
iostream	istream，wistream 从流读取数据 ostream，wostream 向流写入数据 iostream，wiostream 读写流
fstream	ifstream，wifstream 从文件读取数据 ofstream，wofstream 向文件写入数据 fstream，wfstream 读写文件
sstream	istringstream，wistringstream 从 string 读取数据 ostringstream，wostringstream 向 string 写入数据 stringstream，wstringstream 读写 string

  - 概念上，设备类型和字符大小都不会影响我们要执行的IO操作
    - 标准库使我们能忽略这些不同类型的流之间的差异，这是通过继承机制实现的，利用模板我们可以使用具有继承关系的类，而不必了解继承机制张如何工作的细节
  - IO对象无赋值和拷贝
    - 我们不能拷贝或对IO对象赋值，进行IO操作的函数通常以引用的方式传递和返回流，读写一个IO对象会改变其状态，因此传递和返回引用不是const的
  - IO的条件状态
    - IO操作一个与生俱来的问题就是可能发生错误
      - s。 ignore(int n =1, int delim = EOF);：此函数的作用是跳过输入流中的 n 个字符，或跳过 delim 及其之前的所有字符，哪个条件先满足就按哪个执行。两个参数都有默认值，因此 cin.ignore() 就等效于 cin.ignore(1, EOF)， 即跳过一个字符。

表 8.2：IO 库条件状态

strm::iostate	*strm* 是一种 IO 类型，在表 8.1（第 278 页）中已列出。iostate 是一种机器相关的类型，提供了表达条件状态的完整功能
strm::badbit	*strm*::badbit 用来指出流已崩溃
strm::failbit	*strm*::failbit 用来指出一个 IO 操作失败了

strm::eofbit	*strm*::eofbit 用来指出流到达了文件结束
strm::goodbit	*strm*::goodbit 用来指出流未处于错误状态。此值保证为零
s.eof()	若流 s 的 eofbit 置位，则返回 true
s.fail()	若流 s 的 failbit 或 badbit 置位，则返回 true
s.bad()	若流 s 的 badbit 置位，则返回 true
s.good()	若流 s 处于有效状态，则返回 true
s.clear()	将流 s 中所有条件状态位复位，将流的状态设置为有效。返回 void
s.clear(flags)	根据给定的 flags 标志位，将流 s 中对应条件状态位复位。flags 的类型为 *strm*::iostate。返回 void
s.setstate(flags)	根据给定的 flags 标志位，将流 s 中对应条件状态位置位。flags 的类型为 *strm*::iostate。返回 void
s.rdstate()	返回流 s 的当前条件状态，返回值类型为 strm::iostate

- 一个流一旦发生错误，其上后续的IO操作都会失败。只有当一个流处于无错状态的时候，我们才可以从他读取数据，向他写入数据
- 确定一个流对象的状态最简单的方法使将它作为一个条件使用
  while（cin>>word）。。。
- 查询流的状态
  - IO库定义了一个与机器无关的iostate类型，它提供了表达流状态的完整功能。
  - IO库定义了4个iostate类型的constexpr的值，表示特定的位模式这些值用来表示特定类型的IO条件，可以与位运算符一起使用来一次性检测或设置多个标志位
    - badbit表示系统级错误，如不可恢复的读写错误。通常情况下，一旦badbit被置位，流就无法再使用了。
    - 发生可恢复错误后，failbit被置位，如期望读取数值却读出一个字符等错误。这种问题通常是可以修正的，流还可以继续使用。
    - 如果到达文件结束位置，eofbit和failbit都会被置位。goodbit的值为0，表示流未发生错误。
    - 如果badbit、failbit和eofbit任一个被置位，则检测流状态的条件会失败。
  - 查询标志位的状态
    - 操作good在所有错误位均未置位的情况下返回true，而bad、fail和eof则在对应错误位被置位时返回true。此外，在badbit被置位时，fail也会返回true。
    - 这意味着，使用good或fail是确定流的总体状态的正确方法。实际上，我们将流当作条件使用的代码就等价于！fail（）。而eof和bad操作只能表示特定的错误。
- 管理条件状态
  - 流对象的rdstate成员返回一个iostate值，对应流的当前状态。
    - setstate操作将给定条件位置位，表示发生了对应错误。
    - clear成员是一个重载的成员（参见6.4节，第206页）：它有一个不接受参数的版本，而另一个版本接受一个iostate类型的参数。
      - clear不接受参数的版本清除（复位）所有错误标志位。执行clear（）后，调用good会返回true。我们可以这样使用这些成员：

```
// 记住 cin 的当前状态
auto old_state = cin.rdstate();  // 记住 cin 的当前状态
cin.clear();                     // 使 cin 有效
process_input(cin);              // 使用 cin
cin.setstate(old_state);         // 将 cin 置为原有状态
```

- 带参数的clear版本接受一个iostate值，表示流的新状态。为了复位单一的条件状态位，我们首先用rdstate读出当前条件状态，然后用位操作将所需位复位来生成新的状态。例如，下面的代码将failbit和badbit复位，但保持eofbit不变：

```
// 复位 failbit 和 badbit，保持其他标志位不变
cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit);
```

- 管理输出缓冲（控制台）
  - 每个输出流都管理一个缓冲区，用来保存程序读写的数据。
  - 文本串可能立即打印出来，也有可能被操作系统保存在缓冲区中，随后在进行打印
  - 允许操作系统将多个输出操作组合为单一的设备写操作可以带来很大的性能提升
  - 导致缓冲刷新（即数据真正写道输出设备或文件）的原因有很多
    - 程序正常结束，作为main函数的return操作的一部分，刷新缓冲
    - 缓冲区满的时候刷新缓冲，而后新的数据才能继续写入缓冲区
    - 我们可以使用endl来显式的刷新缓冲区
  - 在每个输出操作之后，我们可以使用操纵符unibuf设置流的内部状态，来清空缓冲区，默认情况下，对cerr时设置unibuf，因此写道cerr的内容都是立刻刷新的
  - 一个输出流可能被关联到另一个流，在这种情况下，到读写被关联的流时，关联到的流的缓冲区会被刷新。
    cin和cerr都关联到cout，因此读写cin或写cerr都会导致cout的缓冲区被刷新
  - 刷新输出缓冲区
    - endl，flush，ends的操作

```
cout << "hi!" << endl;   // 输出 hi 和一个换行，然后刷新缓冲区
cout << "hi!" << flush;  // 输出 hi，然后刷新缓冲区，不附加任何额外字符
cout << "hi!" << ends;   // 输出 hi 和一个空字符，然后刷新缓冲区
```

  - unitbuf操纵符
    - 如果想在每次输出操作后都刷新缓冲区，我们可以使用unitbuf操纵符，他告诉流在接下来的每次写操作之后都进行一次flush操作
    - nounibuf操纵符则重置流，使其恢复正常的系统管理的缓冲区刷新机制

```
cout << unitbuf;          // 所有输出操作后都会立即刷新缓冲区
// 任何输出都立即刷新，无缓冲
cout << nounitbuf;        // 回到正常的缓冲方式
```

  - 关联输入和输出流
    - 当一个输入流被关联到一个输出流时，任何试图从输入流读取数据的操作都会先刷新关联的输出流，标准库将cin和cout关联在一起
  - 注意
    - 如果程序异常终止，输入缓冲区是不会被刷新的，当一个程序崩溃后，它所输出的数据很可能停留在输出缓冲区中等待打印
    - 当调试一个已经崩溃的程序时，需要确认那些你认为已经输出的数据确实已经刷新了。
- 文件的输入输出
  - 头文件fstream定义了三个类型来支持文件IO

- ifstream从一个给定的文件读取数据
- ofstream向一个给定文件写入数据
- fstream可以读写给定文件

表 8.3：**fstream** 特有的操作

fstream fstrm;	创建一个未绑定的文件流。*fstream* 是头文件 fstream 中定义的一个类型
fstream fstrm(s);	创建一个 *fstream*，并打开名为 s 的文件。s 可以是 string 类型，或者是一个指向 C 风格字符串的指针（参见 3.5.4 节，第 109 页）。这些构造函数都是 explicit 的（参见 7.5.4 节，第 265 页）。默认的文件模式 mode 依赖于 *fstream* 的类型
fstream fstrm(s, mode);	与前一个构造函数类似，但按指定 mode 打开文件
fstrm.open(s)	打开名为 s 的文件，并将文件与 fstrm 绑定。s 可以是一个 string 或一个指向 C 风格字符串的指针。默认的文件 mode 依赖于 *fstream* 的类型。返回 void
fstrm.close()	关闭与 fstrm 绑定的文件。返回 void
fstrm.is_open()	返回一个 bool 值，指出与 fstrm 关联的文件是否成功打开且尚未关闭

- 这些类型提供的操作与我们之前已经使用过的对象cin和cout的操作一样，我们可以用IO运算符来读写文件，可以用getline从一个ifstream读取数据

- 使用文件流对象
  - 当我们想要读写一个文件时，可以定义一个文件流对象，并将对象与文件关联起来，每个文件流类都定义了一个名为open的成员函数，它完成一些系统相关操作，来定位给定的文件，并视情况打开为读写模式

```
ifstream in(ifile);      // 构造一个 ifstream 并打开给定文件
ofstream out;            // 输出文件流未关联到任何文件
```

  - 关联文件流对象
    - 我们使用tie函数来关联文件的使用流
    - tie有两个重载的版本
      - 一个版本不带参数，返回指向输出流的指针
        auto p=cin.tie():此时返回std：：ostream的指针
      - tie的第二个版本接受一个指向ostream的指针，然后把对于的流关联起来
        cin.tie(&cout):把cin和cout关联起来
      - 如果不想关联，可以使用nullptr
        cin.tie(nullptr)
  - 成员函数open和close
    - 如果我们定义了一个空文件流的对象可以随后调用open来将它与文件关联起来
      - 如果我们创建了一个文件流，可以直接在后面跟上括号附上要打开的文件，也可以先创建一个空的文件流，再用open进行文件的打开

```
ifstream in(ifile);          // 构筑一个 ifstream 并打开给定文件
ofstream out;                // 输出文件流未与任何文件相关联
out.open(ifile + ".copy");   // 打开指定文件
```

    - 我们可以利用条件判断是否open成功【if（out）】
    - 一旦一个文件流被打开，它就保持与对应文件的关联，对一个已经打开的文件流调用open失败并且会导致failbit置位

- 为了将文件流关联到另外一个文件，必须首先关闭已经关联的文件。一旦文件关闭成功，我们可以打开新的文件

```
in.close();                 // 关闭文件
in.open(ifile + "2");       // 打开另一个文件
```

- 当一个fstream对象被销毁时，close会自动被调用

□ 文件模式

- 每个流都关联有一个文件模式，用来指出如何使用文件

表 8.4：文件模式

in	以读方式打开
out	以写方式打开
app	每次写操作前均定位到文件末尾
ate	打开文件后立即定位到文件末尾
trunc	截断文件
binary	以二进制方式进行 IO

- 无论用哪种方式打开文件，我们都可以指定文件模式，调用open打开文件时可以，用一个文件名初始化流来隐式的打开文件时也可以
- 指定文件模式有如下限制
  - ◇ 只可以对ofstream或fstream对象设定为out模式
  - ◇ 只可以对ifstream或fstream对象设定为in模式
  - ◇ 只有当out也被设定的时候才可以设定trunc模式
  - ◇ 只要trunc没被设定，就可以设定app模式，在app模式下，即使没有显式的指定为out模式，文件也总是以输出的方式打开
  - ◇ 默认情况下，即使我们没有指定trunc，以out模式打开的文件也会被截断。
    - ▸ 为了保留以out模式打开的文件的内容，我们必须同时指定app模式，
    - ▸ 或者同时指定in模式，即打开文件的同时进行读写操作
  - ◇ ate和binary模式可用于任何类型的文件流对象，且可以与其他任何文件模式组合使用
  - ◇ 每个文件流类型都定义了一个默认的文件模式，当我们未指定文件模式时，就使用此默认模式。
    - ▸ 与ifstream关联的文件默认以in模式打开
    - ▸ 与ofstream关联的文件默认以out模式打开
    - ▸ fstream关联的文件默认以in和out模式打开
  - ◇ 默认情况下，当我们打开一个ofstream时，文件的内容会被丢弃，阻止一个ofstream清空给定文件内容的方法是同时指定app模式
    - ▸ 保留ofstream打开的文件中已有数据的唯一办法式显式指定app或in模式

```
// 在这几条语句中，file1 都被截断
ofstream out("file1"); // 隐含以输出模式打开文件并截断文件
ofstream out2("file1", ofstream::out); // 隐含地截断文件
ofstream out3("file1", ofstream::out | ofstream::trunc);
// 为了保留文件内容，我们必须显式指定 app 模式
ofstream app("file2", ofstream::app); // 隐含为输出模式
ofstream app2("file2", ofstream::out | ofstream::app);
```

- ◇ 每次调用open时都会确定文件模式
  - ▸ 对于一个给定流，每当打开文件的时候，都可以改变其文件格式

```
ofstream out; // 未指定文件打开模式
out.open("scratchpad"); // 模式隐含设置为输出和截断
out.close(); // 关闭 out，以便我们将其用于其他文件
out.open("precious", ofstream::app); // 模式为输出和追加
out.close();
```

  - ▸ 在每次打开文件时，都要设置文件模式，可能是显式的设置，也可能是隐式的设置。当程序未指定模式时，就使用默认值

- □ string流
  - ◆ sstream头文件定义了三个类型来支持内存io，这些类型可以向string写入数据，从string读取数据，就像string是一个io流一样
  - ◆ istringstream从string读取数据，ostringstream向string写入数据，而头文件stringstream既可以从string读数据也可向string写数据。
  - ◆ 与fstream类型相似，头文件sstream中定义的类型都继承自我们已经使用过的iostream的类型
  - ◆ sstream中定义的类型还增加了一些成员来管理与流相关联的string，可以对stringstream对象调用这些操作，但是不能对其他io类型调用这些操作

表 8.5：stringstream 特有的操作

sstream strm;	strm 是一个未绑定的 stringstream 对象。*sstream* 是头文件 sstream 中定义的一个类型
sstream strm(s);	strm 是一个 *sstream* 对象，保存 string s 的一个拷贝。此构造函数是 explicit 的（参见 7.5.4 节，第 265 页）
strm.str()	返回 strm 所保存的 string 的拷贝
strm.str(s)	将 string s 拷贝到 strm 中。返回 void

  - ◆ 注意open是打开文件，不是调用流
  - ◆ 使用istringstream
    - ◇ 当我们的某些工作是对整行文本进行处理，而其他一些工作是处理行内的单个单词的时候，通常可以使用istringstream

- □ 注意
  - ◆ 每个循环进行的时候会默认销毁流，在不销毁流的情况下要对流进行clear操作后才能使用

- 小结
  - C++使用标准库类来处理面向流的输入和输出：
  - iostream处理控制台IO·
  - fstream处理命名文件IO·
  - stringstream完成内存string的IO类

- fstream和stringstream都是继承自类iostream的。输入类都继承自istream，输出类都继承自ostream。因此，可以在istream对象上执行的操作，也可在ifstream或istringstream对象上执行。继承自ostream的输出类也有类似情况。
- 每个IO对象都维护一组条件状态，用来指出此对象上是否可以进行IO操作。如果遇到了错误——例如在输入流上遇到了文件末尾，则对象的状态变为失效，所有后续输入操作都不能执行，直至错误被纠正。标准库提供了一组函数，用来设置和检测这些状态。